МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра САПР

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Компьютерная графика»

Тема: Исследование алгоритмов выявления видимости сложных сцен

Студенты гр. 1302 _____ Марзаева В.И.

_____ Новиков Г.В.

_____ Романова О.В.

Преподаватель _____ Колев Г.Ю.

Санкт-Петербург

2024

Цель работы

Обеспечить реализацию раскраски поверхностей, сформированных при выполнении работы по теме 3. При этом лицевая и оборотная сторона должны окрашиваться в разные цвета. Обеспечить поворот окрашенной поверхности вокруг осей X и Y.

Теоретическая часть программы

Билинейные поверхности – простейшие трехмерные поверхности. Они задаются на ограниченном участке с заданием в пространстве 4-х угловых точек поверхности. Уравнение билинейчатой поверхности представляется как:

$$\bar{Q}(u, w) = \bar{P}_{00}(1-u)(1-w) + \bar{P}_{01}(1-u)w + \bar{P}_{10}u(1-w) + \bar{P}_{11}uw$$

$$0 \le u \le 1$$

$$0 \le w \le 1$$

Если u=0; w=0, то попадаем в точку $\bar{P}_{00} = \bar{Q}(u,w)$

Если u=1; w=0, то попадаем в точку $\bar{P}_{10} = \bar{Q}(u, w)$

Если u=1; w=1, то попадаем в точку $\bar{P}_{11} = \bar{Q}(u, w)$

Вращение осуществляется вокруг осей X и Y относительно наблюдателя (то есть при вращении поверхности оси не вращаются). Для реализации этого в памяти хранится матрица поворота. Это позволяет совершать поворот вокруг осей X, Y, Z поверхности и возвращать поверхность в исходное положение. Углы при выводе на экран вычисляются из матрицы.

Для того, чтобы закрашенная поверхность правильно отображалась, необходимо определить видимость каждого пикселя. Для этого используется алгоритм, основанный на алгоритме выявления видимых частей сцены, с использованием Z-буфера.

Закрашивание происходит следующим образом: в Z-буфер записываются значения расстояния от наблюдателя для каждого пикселя на экране. В буфер цветов пикселей framebuffer для каждого пикселя на экране записывается индекс цвета фона. Для каждого полигона поверхности берутся точки на плоскости (экране), лежащие внутри проекции этого полигона. Для каждой точки находится Z (на основании уравнения плоскости, составленного по 2 прямым). Это значение сравнивается со значением в Z-буфере по координатам точки, если значение больше, то Z текущей точки записывается в буфер, и меняется индекс цвета в framebuffer в зависимости от того, какой стороной повернут полигон. Сторона полигона определяется на основе расположения его вершин. Если вершины расположены по часовой стрелке, это передняя сторона, если против часовой – задняя. Если координаты точки совпадают с координатой одной из вершин полигона, цвет меняется на цвет вершины.

Пример работы программы

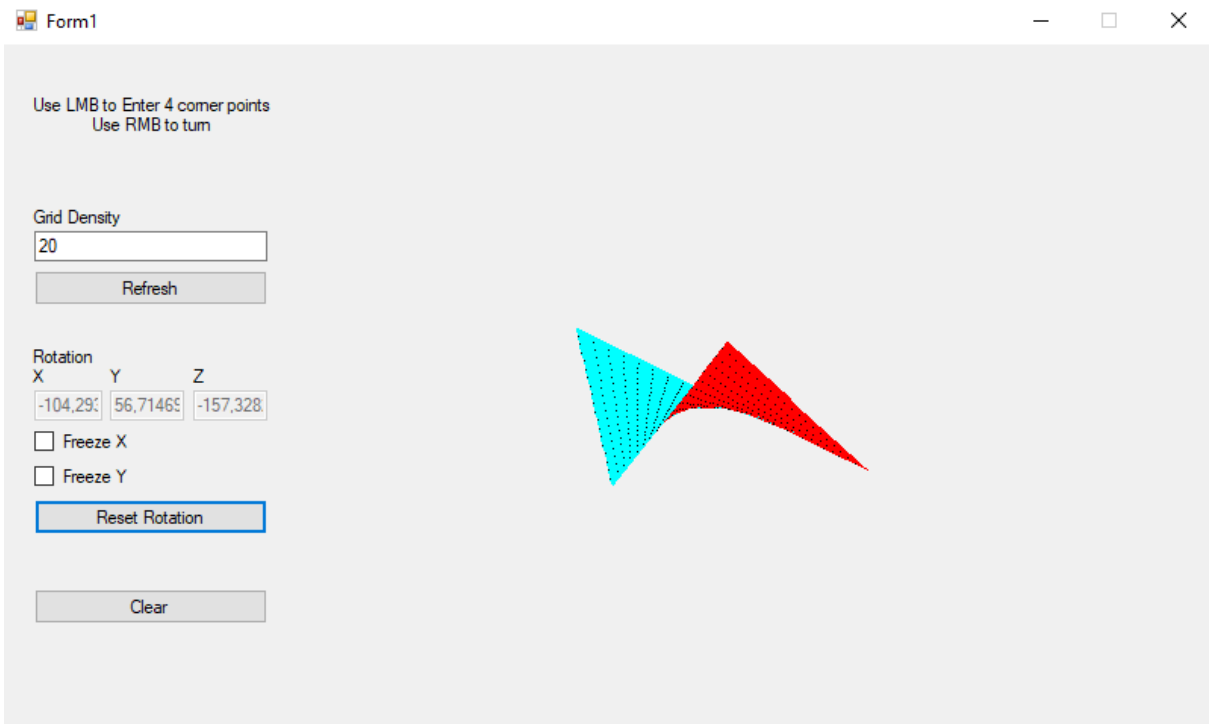

Рис. 1. Поверхность 1 вид 1

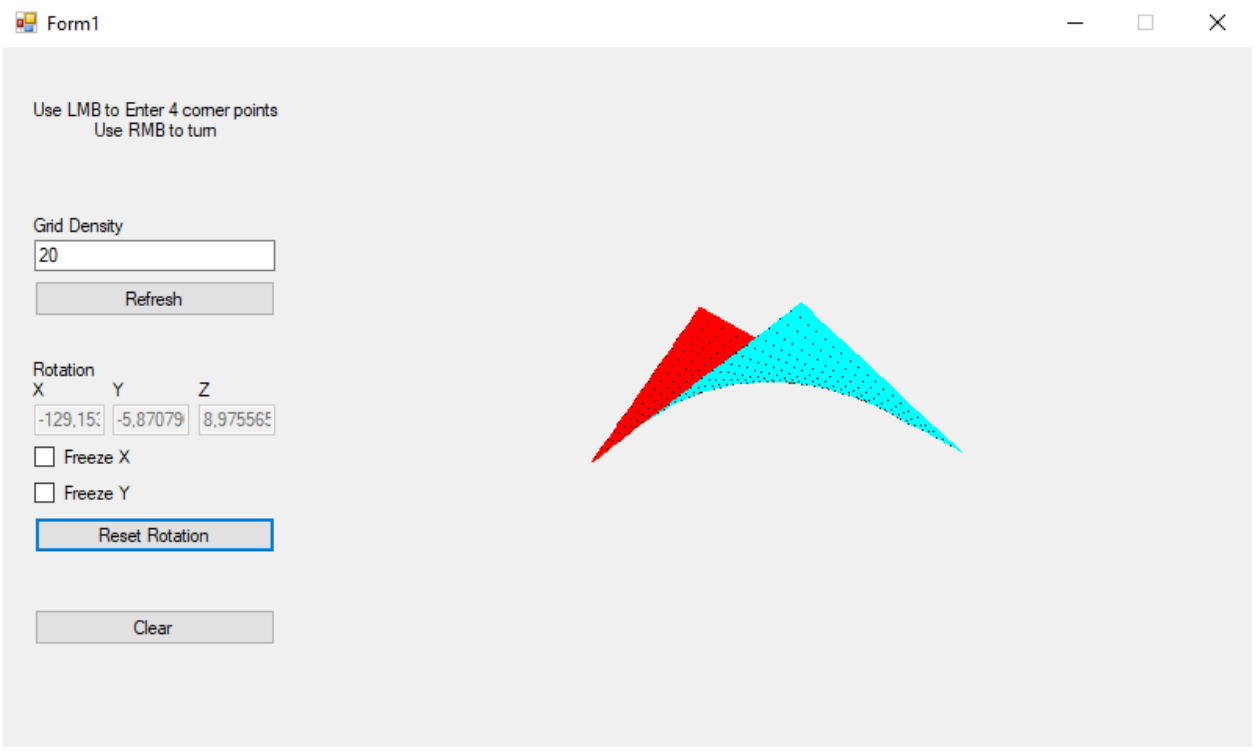

Рис. 2. Поверхность 1 вид 2

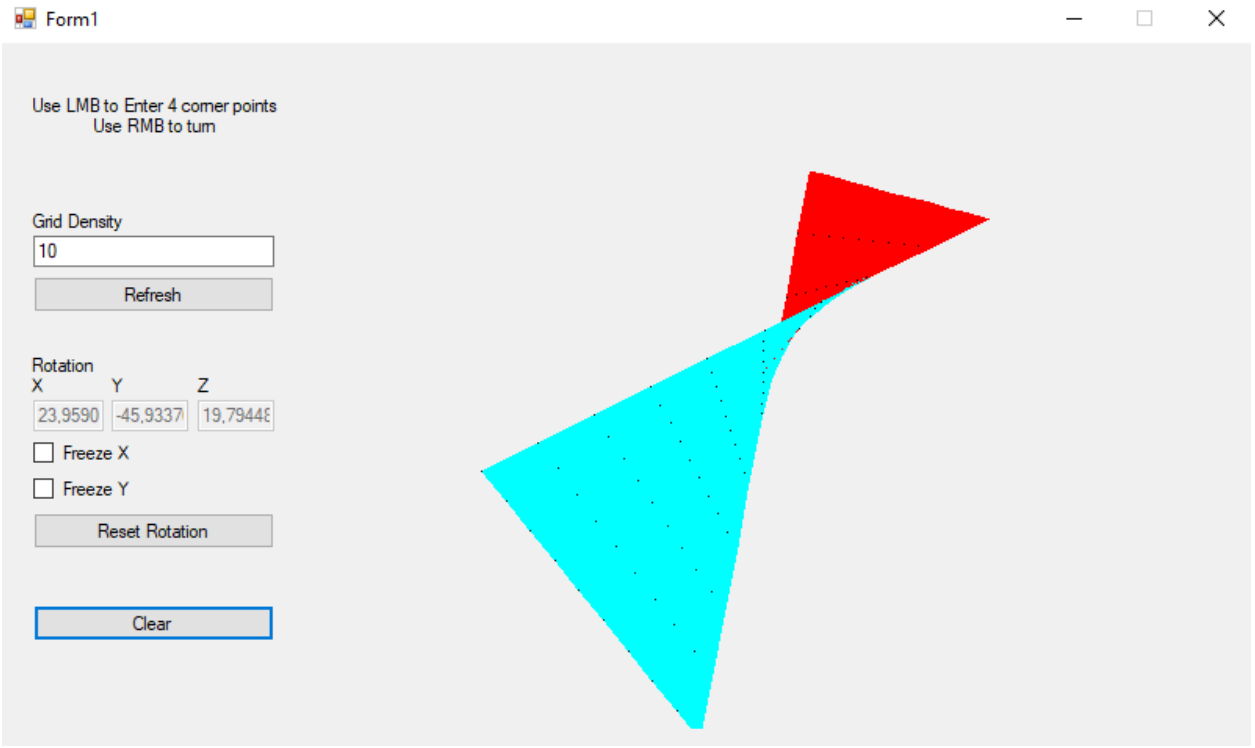

Рис. 3. Поверхность 2 вид 1

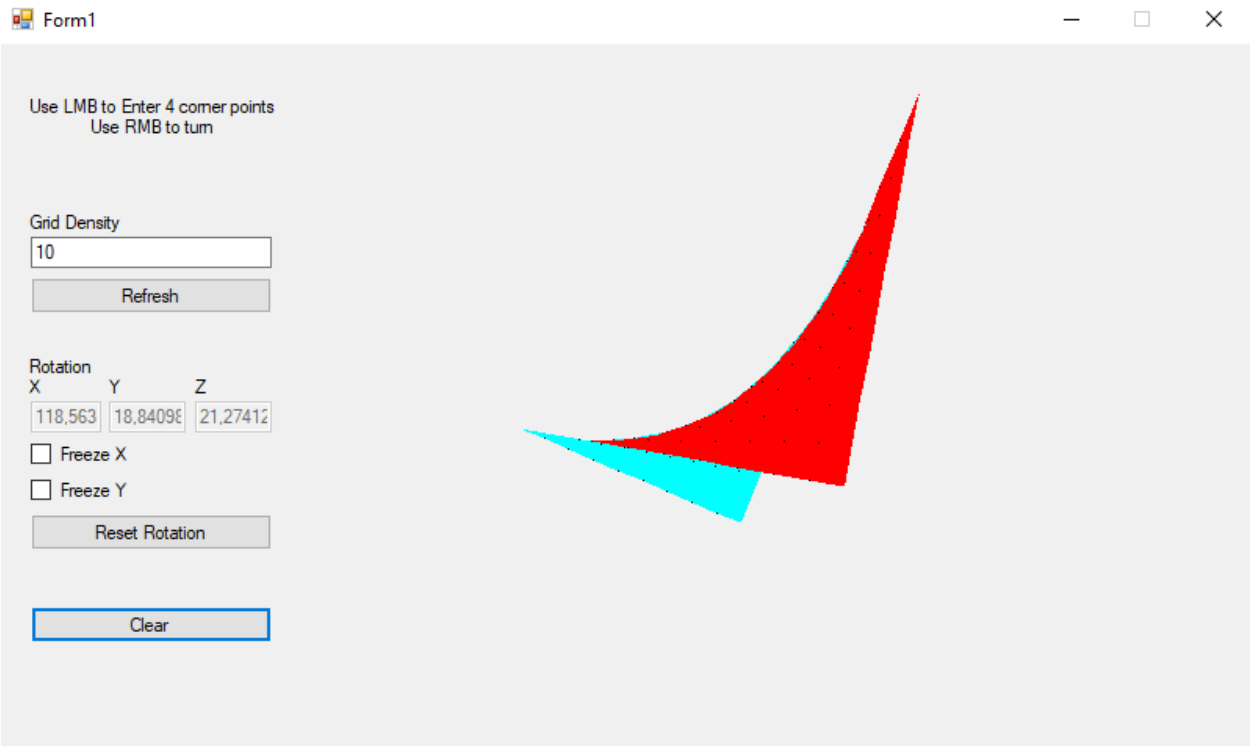

Рис. 4. Поверхность 2 вид 2

Код программы

Form1.cs:

```
using System;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;

using MathNet.Numerics.LinearAlgebra;

namespace lab5
{
    public partial class Form1 : Form
    {
        private Point3D center = new Point3D(0, 0, 0);
        private BilinearSurface surface = new BilinearSurface();

        private bool rightMousePressed = false;
        private Point mouseDownPoint = new Point(0, 0);
        private Vector<double> deltaRotation = Vector<double>.Build.Dense(2);

        private bool cursorHidden = false;
        private Point cursorFixPosition;

        private int usualPointSize = 2;
        private int cornerPointSize = 5;
        private double sensitivityX = 0.01;
        private double sensitivityY = 0.01;

        private Brush surfaceFrontBrush = Brushes.Cyan;
        private Brush surfaceBackBrush = Brushes.Red;
        private Brush cornerBrush = Brushes.Black;
        private Brush pointBrush = Brushes.Black;
```

```
public int GridDensity
{
    get { return int.TryParse(textBox1.Text, out int density) ? density : 0; }
}

public bool FreezeX
{
    get { return checkBox1.Checked; }
}

public bool FreezeY
{
    get { return checkBox2.Checked; }
}

public Form1()
{
    InitializeComponent();
    textBox1.Text = "20";
    center[0] = pictureBox1.Width / 2;
    center[1] = pictureBox1.Height / 2;

    surface.cornerBrush = cornerBrush;
    surface.pointBrush = pointBrush;
    surface.surfaceFrontBrush = surfaceFrontBrush;
    surface.surfaceBackBrush = surfaceBackBrush;
    surface.pointFatness = usualPointSize;
    surface.cornerFatness = cornerPointSize;
}

private void PictureBox1_Paint(object sender, PaintEventArgs e)
{
    if (surface.Corners.Count == 4 && GridDensity != 0)
    {
        surface.Fill(e.Graphics, pictureBox1);
    }
    else
    {
        surface.DrawCornerPoints(e.Graphics);
    }

    Vector<double> angles = surface.Rotation.GetAngles();
    textBox2.Text = (angles[0] * 180 / Math.PI).ToString();
    textBox3.Text = (angles[1] * 180 / Math.PI).ToString();
    textBox4.Text = (angles[2] * 180 / Math.PI).ToString();
    DisplayMessage();
}

private void PictureBox1_MouseClick(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left && surface.Corners.Count != 4)
    {
        Point3D point = new Point3D(e.Location.X, e.Location.Y, 0d);
        surface.Corners.Add(point);

        if (surface.Corners.Count == 4)
        {
            surface.Calculate(GridDensity);
        }
        pictureBox1.Refresh();
    }
}

private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
    if (rightMousePressed)
    {
        if (!FreezeX)
        {
            deltaRotation[0] = -(e.Y - mouseDownPoint.Y) * sensitivityX;
```

```
        }

        if (!FreezeY)
        {
            deltaRotation[1] = (e.X - mouseDownPoint.X) * sensitivityY;
        }

        surface.Rotate(center, deltaRotation);
        deltaRotation.Clear();

        pictureBox1.Refresh();
        Cursor.Position = cursorFixPosition;
    }
}

private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)
    {
        rightMousePressed = true;
        mouseDownPoint.X = e.X;
        mouseDownPoint.Y = e.Y;
        if (!cursorHidden)
        {
            Cursor.Hide();
            cursorHidden = true;
            cursorFixPosition = Cursor.Position;
        }
    }
}

private void pictureBox1_MouseUp(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)
    {
        rightMousePressed = false;
        if (cursorHidden)
        {
            Cursor.Show();
            cursorHidden = false;
        }
    }
}

private void button1_Click(object sender, EventArgs e)
{
    surface.ResetRotation(center);
    pictureBox1.Refresh();
}

private void button2_Click(object sender, EventArgs e)
{
    surface.Calculate(GridDensity);
    pictureBox1.Refresh();
}

private void Button3_Click(object sender, EventArgs e)
{
    ClearForm();
}

private void ClearForm()
{
    surface.Polygons.Clear();
    surface.PointArray.Clear();
    surface.Corners.Clear();
    surface.Rotation.Clear();
    pictureBox1.Refresh();
}
```

```
        private void DisplayMessage()
        {
            if (GridDensity <= 1)
            {
                label2.ForeColor = Color.Red;
                label2.Text = "Invalid grid density";
            }
            else
            {
                label2.ForeColor = Color.Black;
                label2.Text = "Use LMB to Enter 4 corner points\nUse RMB to turn";
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            pictureBox1.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
        }
    }
}
```

BilinearSurface.cs:

```
using MathNet.Numerics.LinearAlgebra;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace lab5
{
    internal class BilinearSurface
    {
        public List<Polygon> Polygons { get; }
        public List<Point3D> PointArray { get; }
        public List<Point3D> Corners { get; }
        public Rotation Rotation { get; }

        public Brush pointBrush = Brushes.Black;
        public Brush cornerBrush = Brushes.Black;
        public Brush surfaceFrontBrush = Brushes.Cyan;
        public Brush surfaceBackBrush = Brushes.Cyan;
        public Pen polygonBorderPen = Pens.Black;

        public int pointFatness = 1;
        public int cornerFatness = 4;

        private enum PixelType { Point, SurfaceFront, SurfaceBack, Background }

        public BilinearSurface()
        {
            Polygons = new List<Polygon>();
            PointArray = new List<Point3D>();
            Corners = new List<Point3D>();
            Rotation = new Rotation();
        }

        public void Fill(Graphics g, PictureBox pictureBox)
        {
            double[,] zBuffer = new double[pictureBox.Height, pictureBox.Width];
            PixelType[,] frameBuffer = new PixelType[pictureBox.Height, pictureBox.Width];
            FillBuffers(ref zBuffer, ref frameBuffer, pictureBox);
            DrawPixels(g, frameBuffer);
        }

        private void FillBuffers(ref double[,] zBuffer, ref PixelType[,] frameBuffer, PictureBox pictureBox)
        {
            for (int y = 0; y < zBuffer.GetLength(0); y++)
```

```
    {
        for (int x = 0; x < zBuffer.GetLength(1); x++)
        {
            zBuffer[y, x] = double.NegativeInfinity;
            frameBuffer[y, x] = PixelType.Background;
        }
    }

    foreach (Polygon polygon in Polygons)
    {
        List<Point> points = polygon.Get2DPointsInsidePolygon(PointArray);
        Point[] corners = polygon.GetCorners(PointArray).Select(c => c.ToPoint()).ToArray();
        foreach (Point point in points)
        {
            if (!(point.X < 0 || point.X >= pictureBox.Width || point.Y < 0 || point.Y >= pictureBox.Height))
            {
                double z = polygon.GetZValue(PointArray, point);
                if (z > zBuffer[point.Y, point.X])
                {
                    zBuffer[point.Y, point.X] = z;

                    if (polygon.CornersArrangedClockwise(PointArray))
                    {
                        frameBuffer[point.Y, point.X] = PixelType.SurfaceFront;
                    }
                    else
                    {
                        frameBuffer[point.Y, point.X] = PixelType.SurfaceBack;
                    }

                    foreach (Point corner in corners)
                    {
                        if (corner.X == point.X && corner.Y == point.Y)
                        {
                            frameBuffer[point.Y, point.X] = PixelType.Point;
                            break;
                        }
                    }
                }
            }
        }
    }
}

private void DrawPixels(Graphics g, PixelType[,] frameBuffer)
{
    for (int y = 0; y < frameBuffer.GetLength(0); y++)
    {
        for (int x = 0; x < frameBuffer.GetLength(1); x++)
        {
            if (frameBuffer[y, x] == PixelType.Point)
            {
                g.FillRectangle(pointBrush, x, y, 1, 1);
            }
            else if (frameBuffer[y, x] == PixelType.SurfaceFront)
            {
                g.FillRectangle(surfaceFrontBrush, x, y, 1, 1);
            }
            else if (frameBuffer[y, x] == PixelType.SurfaceBack)
            {
                g.FillRectangle(surfaceBackBrush, x, y, 1, 1);
            }
        }
    }
}

public void DrawCornerPoints(Graphics g)
{
    foreach (Point3D corner in Corners)
    {
```

```
            corner.Draw(g, cornerBrush, cornerFatness);
        }
    }

    public void Calculate(int gridDensity)
    {
        Polygons.Clear();
        PointArray.Clear();
        double du = 1.0 / (gridDensity / 1 - 1);
        double dw = 1.0 / (gridDensity / 1 - 1);

        for (int i = 0; i < gridDensity; i++)
        {
            double u = i * du;
            for (int j = 0; j < gridDensity; j++)
            {
                double w = j * dw;
                Point3D point = CalculatePointOnSurface(u, w);
                PointArray.Add(point);
            }
        }

        Polygon polygon;
        for (int i = 0; i < gridDensity - 1; i++)
        {
            for (int j = 0; j < gridDensity - 1; j++)
            {
                polygon = new Polygon(new int[] { gridDensity * i + j, gridDensity * i + j + 1, gridDensity * (i + 1) + j });
                Polygons.Add(polygon);
                polygon = new Polygon(new int[] { gridDensity * i + j + 1, gridDensity * (i + 1) + (j + 1), gridDensity * (i + 1) + j
});
                Polygons.Add(polygon);
            }
        }
    }

    private Point3D CalculatePointOnSurface(double u, double w)
    {
        Point3D point = new Point3D();
        if (Corners.Count != 4)
        {
            return point;
        }

        for (int i = 0; i < 3; i++)
        {
            point[i] = Corners[0][i] * (1 - u) * (1 - w) + Corners[1][i] * (1 - u) * w + Corners[2][i] * u * (1 - w) + Corners[3][i] * u
* w;
        }

        return point;
    }

    public void Rotate(Point3D center, Vector<double> rotation)
    {
        Matrix<double> matrixX = GetXRotationMatrix(rotation);
        Matrix<double> matrixY = GetYRotationMatrix(rotation);

        foreach (Point3D point in PointArray)
        {
            point.RotateByMatrix(center, matrixX);
            point.RotateByMatrix(center, matrixY);
        }

        foreach (Point3D corner in Corners)
        {
            corner.RotateByMatrix(center, matrixX);
            corner.RotateByMatrix(center, matrixY);
        }
```

```
            Rotation.Matrix = matrixY * matrixX * Rotation.Matrix;
        }

        public void ResetRotation(Point3D center)
        {
            foreach (Point3D point in PointArray)
            {
                point.RotateByMatrix(center, Rotation.Matrix.Inverse());
            }

            foreach (Point3D corner in Corners)
            {
                corner.RotateByMatrix(center, Rotation.Matrix.Inverse());
            }

            Rotation.Clear();
        }

        private Matrix<double> GetXRotationMatrix(Vector<double> rotation)
        {
            double cos = Math.Cos(rotation[0]);
            double sin = Math.Sin(rotation[0]);

            double[,] rotX = {
                { 1, 0, 0},
                { 0, cos, -sin},
                { 0, sin, cos}
            };

            return Matrix<double>.Build.DenseOfArray(rotX);
        }

        private Matrix<double> GetYRotationMatrix(Vector<double> rotation)
        {
            double cos = Math.Cos(rotation[1]);
            double sin = Math.Sin(rotation[1]);

            double[,] rotY = {
                { cos, 0, sin},
                { 0, 1, 0},
                { -sin, 0, cos}
            };

            return Matrix<double>.Build.DenseOfArray(rotY);
        }

    }
}
```

Polygon.cs:

```
using MathNet.Numerics.LinearAlgebra;
using MathNet.Numerics.LinearAlgebra.Complex;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;

namespace lab5
{
    internal class Polygon
    {
        public int[] PointIndices { get; set; }

        public Polygon(int[] pointIndices)
        {
            if (pointIndices.Length != 3) throw new ArgumentException("Polygon must have 3 vertices");

            PointIndices = new int[3];
            for (int i = 0; i < pointIndices.Length; i++)
            {
```

```
            PointIndices[i] = pointIndices[i];
        }
    }

    public Point3D[] GetCorners(List<Point3D> pointArray)
    {
        Point3D[] points = new Point3D[3];
        points[0] = pointArray[PointIndices[0]];
        points[1] = pointArray[PointIndices[1]];
        points[2] = pointArray[PointIndices[2]];
        return points;
    }

    public double GetZValue(List<Point3D> pointArray, Point point)
    {
        double[] coefficients = GetPlaneEquation(pointArray);
        double z = -(coefficients[0] * point.X + coefficients[1] * point.Y + coefficients[3]) / coefficients[2];
        return z;
    }

    public List<Point> Get2DPointsInsidePolygon(List<Point3D> pointArray)
    {
        List<Point> points = new List<Point>();
        Point[] corners = GetCorners(pointArray).Select((p) => p.ToPoint()).ToArray();

        int maxYCornerIndex = 0;
        int middleYCornerIndex = 1;
        int minYCornerIndex = 2;
        for (int i = 0; i < corners.Length; i++)
        {
            if (corners[i].Y < corners[minYCornerIndex].Y)
            {
                minYCornerIndex = i;
            }
            else if (corners[i].Y > corners[maxYCornerIndex].Y)
            {
                maxYCornerIndex = i;
            }
            middleYCornerIndex = 3 - minYCornerIndex - maxYCornerIndex;
        }
        Point maxYCorner = corners[maxYCornerIndex];
        Point middleYCorner = corners[middleYCornerIndex];
        Point minYCorner = corners[minYCornerIndex];

        for (int y = maxYCorner.Y; y > minYCorner.Y; y--)
        {
            int[] xBorders = GetHorizontalLineXBorders(maxYCorner, middleYCorner, minYCorner, y);
            for (int x = xBorders[0]; x < xBorders[1]; x++)
            {
                points.Add(new Point(x, y));
            }
        }

        return points;
    }

    private int[] GetHorizontalLineXBorders(Point maxYCorner, Point middleYCorner, Point minYCorner, int y)
    {
        int[] borders = new int[2];

        double[] lineCoefficients;
        double k, m;
        if (maxYCorner.X == minYCorner.X)
        {
            borders[1] = maxYCorner.X;
        }
        else
        {
            lineCoefficients = GetLineEquation(maxYCorner, minYCorner);
            k = lineCoefficients[0];
```

```
            m = lineCoefficients[1];
            borders[1] = (int)Math.Round((y - m) / k);
        }

        if (y > middleYCorner.Y)
        {
            if (maxYCorner.X == middleYCorner.X)
            {
                borders[0] = maxYCorner.X;
            }
            else
            {
                lineCoefficients = GetLineEquation(maxYCorner, middleYCorner);
                k = lineCoefficients[0];
                m = lineCoefficients[1];
                borders[0] = (int)Math.Round((y - m) / k);
            }
        }
        else
        {
            if (minYCorner.X == middleYCorner.X)
            {
                borders[0] = minYCorner.X;
            }
            else
            {
                lineCoefficients = GetLineEquation(middleYCorner, minYCorner);
                k = lineCoefficients[0];
                m = lineCoefficients[1];
                borders[0] = (int)Math.Round((y - m) / k);
            }
        }

        return borders.OrderBy(x => x).ToArray();
    }

    public bool CornersArrangedClockwise(List<Point3D> pointArray)
    {
        Point[] points = GetCorners(pointArray).Select(p => p.ToPoint()).ToArray();
        if (points[0].X == points[1].X)
        {
            if (points[0].Y > points[1].Y)
            {
                return points[2].X < points[0].X;
            }
            else
            {
                return points[2].X > points[0].X;
            }
        }

        double[] lineCoefficients = GetLineEquation(points[0], points[1]);
        double k = lineCoefficients[0];
        double m = lineCoefficients[1];

        if (points[0].X < points[1].X)
        {
            return points[2].Y < k * points[2].X + m;
        }
        else
        {
            return points[2].Y > k * points[2].X + m;
        }
    }

    private double[] GetLineEquation(Point point1, Point point2)
    {
        // y = kx + m
        if (point1.X == point2.X) throw new ArgumentException("Method does not work for line equations like x = c");
        double k = (double)(point1.Y - point2.Y) / (point1.X - point2.X);
```

```
            double m = point2.Y - k * point2.X;
            return new double[] { k, m };
        }

        private double[] GetPlaneEquation(List<Point3D> pointArray)
        {
            double[] coefficients = new double[4];
            Point3D[] corners = GetCorners(pointArray);
            Vector<double> v1 = corners[1].Coords - corners[0].Coords;
            Vector<double> v2 = corners[2].Coords - corners[0].Coords;

            Vector<double> normal = Cross(v1, v2);

            coefficients[0] = normal[0];
            coefficients[1] = normal[1];
            coefficients[2] = normal[2];

            Point3D point = corners[0];
            coefficients[3] = -(point[0] * coefficients[0] + point[1] * coefficients[1] + point[2] * coefficients[2]);

            return coefficients;
        }

        private Vector<double> Cross(Vector<double> v1, Vector<double> v2)
        {
            Vector<double> result = Vector<double>.Build.Dense(3);
            result[0] = v1[1] * v2[2] - v1[2] * v2[1];
            result[1] = v1[2] * v2[0] - v1[0] * v2[2];
            result[2] = v1[0] * v2[1] - v1[1] * v2[0];
            return result;
        }
    }
}
```

Point3D.cs:

```
using MathNet.Numerics.LinearAlgebra;
using System;
using System.Drawing;

namespace lab5
{
    internal class Point3D
    {
        public Vector<double> Coords { get; set; }

        public Point3D()
        {
            Coords = Vector<double>.Build.Dense(3);
        }

        public Point3D(Vector<double> coords)
        {
            coords.CopyTo(Coords);
        }

        public Point3D(double x, double y, double z)
        {
            Coords = Vector<double>.Build.DenseOfArray(new double[] { x, y, z });
        }

        public Point3D(int x, int y, int z)
        {
            Coords = Vector<double>.Build.DenseOfArray(new double[] { x, y, z });
        }

        public double this[int i]
        {
            get { return Coords[i]; }
            set { Coords[i] = value; }
        }
```

```
        public static Point3D operator +(Point3D point1, Point3D point2)
        {
            return new Point3D(point1.Coords + point2.Coords);
        }

        public static Point3D operator -(Point3D point1, Point3D point2)
        {
            return new Point3D(point1.Coords - point2.Coords);
        }

        public Point ToPoint()
        {
            Point point = new Point();
            point.X = (int)Coords[0];
            point.Y = (int)Coords[1];

            return point;
        }

        public void Draw(Graphics g, Brush brush, int fatness)
        {
            Point point = ToPoint();
            g.FillRectangle(brush, point.X - fatness / 2, point.Y - fatness / 2, fatness, fatness);
        }

        public void RotateByMatrix(Point3D center, Matrix<double> matrix)
        {
            Coords -= center.Coords;
            Coords = matrix * Coords;
            Coords += center.Coords;
        }
    }
}
```

Rotation.cs:

```
using MathNet.Numerics.LinearAlgebra;
using System;

namespace lab5
{
    internal class Rotation
    {
        public Matrix<double> Matrix { get; set; }

        public Rotation()
        {
            Matrix = Matrix<double>.Build.DenseIdentity(3, 3);  // identity matrix (zero rotation)
        }

        public static Rotation operator -(Rotation rotation)
        {
            Rotation newRotation = new Rotation();
            newRotation.Matrix = rotation.Matrix.Inverse();
            return newRotation;
        }

        public Vector<double> GetAngles()
        {
            Vector<double> angles = Vector<double>.Build.Dense(3);
            if (Math.Abs(Matrix[2, 0]) != 1)
            {
                angles[1] = Math.Asin(-Matrix[2, 0]);
                double cosY = Math.Cos(angles[1]);
                angles[0] = Math.Atan2(Matrix[2, 1] / cosY, Matrix[2, 2] / cosY);
                angles[2] = Math.Atan2(Matrix[1, 0] / cosY, Matrix[0, 0] / cosY);
            }
            else
            {
                angles[2] = 0;
```

```
            if (Matrix[2, 0] == -1)
            {
                angles[1] = Math.PI;
                angles[0] = angles[2] + Math.Atan2(Matrix[0, 1], Matrix[0, 2]);
            }
            else
            {
                angles[1] = -Math.PI;
                angles[0] = -angles[2] + Math.Atan2(-Matrix[0, 1], -Matrix[0, 2]);
            }
        }
        return angles;
    }

    public void Clear()
    {
        Matrix = Matrix<double>.Build.DenseIdentity(3, 3);
    }
  }
}
```

Выводы

В данной работе с помощью Windows Forms на C# была реализована программа, которая строит билинейную поверхность по заданным пользователем четырем угловым точкам, определяет видимость точек и закрашивает разные стороны поверхности разными цветами. Также был реализован поворот относительно осей X и Y, с использованием возможности «замораживания» x и y.